ESPRES: Easy Scheduling and Prioritization for SDN

Peter Perešíni[†] Maciej Kuźniar[†] Marco Canini[*] Dejan Kostić[‡]
[†] *EPFL* [*] *Université catholique de Louvain* [‡] *Institute IMDEA Networks*

Network state is always in flux. Due to traffic engineering, topology changes, policy updates, VM migrations, etc., today's networks undergo a variety of large updates that concurrently affect many switches. Transitioning between network states can be a source of instability, leading to outages, disruptions and security vulnerabilities. Consistent network updates [7] introduces a mechanism that guarantees to preserve well defined behaviors when transitioning between states. However, a major problem for this technique is the update performance, that is, the time it takes to install a network state update onto the data-plane—the current generation of OpenFlow switches can install flows with rate as low as 40 rules/second [2].[1] Even moderate-sized updates can take several seconds, during which operators are in the dark about how badly links could be congested. [5] Therefore it is desirable to complete updates quickly. However, we note that the lowest bound of the total time to complete the update is determined by the switch that is last to complete.

We observe that a large *network update* may consist of a set of *sub-updates* that are independent and can be installed in parallel in any order. Each *sub-update* comprises a list of rule modification commands across multiple switches along with an associated dependency graph between commands, as defined in [6]. Our insight is that it is possible to exploit this independence to improve the update performance by carefully managing the scheduling of deciding which *sub-update* to install in which order and, within each *sub-update*, which commands to enqueue at each switch. We can define the update performance to optimize for various objectives, such as:

Finishing sub-updates quickly. While the overall *network update* time is dictated by the slowest switch, we can increase the update efficiency by finishing particular *sub-updates* soon after the update starts. This is especially important for consistent updates [7] since traffic is processed according to a *sub-update* only when the corresponding two-phase commit ends.

Minimizing flow table overhead. Besides performance, an important problem is the rule-space overhead due to the extra number of rules that must be kept at the switches to implement the update. Like others [3, 6], we observe that there is a trade-off between consistency, update speed and number of rules installed at a switch during an update. However, if an update consists of many independent rule commands, a scheduler can reorder the modifications to reduce the rule overhead during an update without impacting the total update time, for example by interleaving rule insertions with rule deletion commands.

Minimizing transient inconsistencies. In the absence of a consistent network update, a *sub-update* may disrupt traffic during the period between the first and last data-plane modifications take place (for example an update that first deletes old rules and then installs new ones). Therefore, it is desirable to keep this transient phase as short as possible by scheduling the last modification to occur soon after the first.

Prioritizing important updates. Certain *sub-updates* or entire *network updates* may be more important than others (*e.g.*, an update blocking all flows from a virus-infected machine should have priority over load shifting due to traffic engineering). As such, they should be applied as early as possible, even if they were issued at a later time than the less important ones.

However, finding the optimal schedule is time consuming and computationally intensive. Further, even if a complete list of individual rule modifications is known beforehand along with their dependency graphs, computing the optimal update schedule is difficult because switches process commands at variable speeds and can queue them up.

To overcome these challenges, we design ESPRES, a layer that operates at runtime by rate-limiting and reordering updates to fully utilize switches without overloading them. We find that this is sufficient to reach the aforementioned goals. Further, combining switch queue management with even simple and quick heuristic scheduling disciplines can significantly improve the update performance. Our early results show that compared to using no scheduler, a simple scheduler yields 4 times quicker *sub-update* completion time for 20th percentile of *sub-updates* and 40% quicker for 50th percentile. Moreover, a scheduling algorithm optimized for rule-space overhead causes only 10% overhead instead of 76% without any scheduler.

ESPRES

ESPRES is a *sub-update* scheduler that runs as a middle layer placed between the controller and the (OpenFlow) switches. As an input, ESPRES receives a stream of updates, each of which is a set of independent *sub-updates*. A *sub-update* consists of (i) rule modifications and their dependency graphs,[2] (ii) a slice definition, that is, a collection of predicates on packet headers and a set of switches and ports this *sub-update* applies to, and (iii) an optional update priority. A dependency between *sub-updates* of different updates exists in case there

[1] While next generation hardware may alleviate the issue, others [1] have argued that certain hardware limitations will stay for several generations of ASIC design. Moreover, as flow table sizes increase, certain updates may affect an even greater number of fine-grained rules.

[2] We assume that these dependencies come either from the controller or the consistent update runtime. In the latter case the dependencies express the different stages of the two-phase commit. Deriving them in an automatic way is part of future work.

is an overlap between their respective slice definitions (there is a nonempty intersection of a set of switches and ports and the conjunction of predicates is nonempty).

Queue management and intra-update scheduling. A key insight in ESPRES is maintaining good responsiveness by actively managing switch command queues. That is, instead of sending (and queuing) all commands at once to a switch (with no possible future reordering/cancellation), ESPRES queues these commands inside the controller and sends to the switch a small yet large enough number to obtain full switch performance. We show the importance of such queue management in Fig. 1—naively sending all available commands to the switch fills up its queue, which may delay the installation of rules for an almost completed *sub-update*. Instead, when the queue length is actively managed, the controller can decide which commands are to be sent next according to a particular scheduling disciplines (*e.g.*, prefer rules from *sub-updates* that already started) and/or different priorities. Additionally, keeping bounded command queues on switches helps avoiding switch error messages that may happen due to the switch being overloaded.

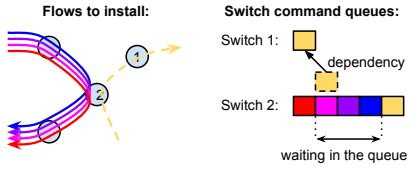

Figure 1: Dashed sub-update is held back by the dependency waiting in switch 2's command queue.

In our current prototype we developed a simple heuristic that optimizes for *sub-update* completion time. Our algorithm prioritizes *sub-updates* according to the number of remaining rule modifications that each *sub-update* contains (*i.e.*, small updates first). Given a selected *sub-update*, the algorithm installs all rule modifications that have their dependencies satisfied if the queue lengths at all affected switches are below a threshold (2 in our experiments).[3]

Inter-update scheduling. As mentioned, different *sub-updates* may have different priorities; ESPRES performs inter-update scheduling to satisfy them. The key idea is to exploit the independence of *sub-updates* associated with different slices, and thus potentially reorder them arbitrarily. Developing an algorithm for this type of scheduling is part of our ongoing work. Besides reordering updates, we envision that, during high churn of updates, the algorithm may defer installing lower priority *sub-updates* and potentially merge them with subsequent *sub-updates* associated with the same slice.

Early results

We used ESPRES to control the installation of a 1000-*sub-updates* update in an IBM topology [4] containing 18 switches in a Mininet environment. The emulator uses the reference OpenFlow switch implementation rate limited to 40 rule modifications/second to correspond to existing hardware switches [2]. Fig. 2 shows that significant benefits come even from simple scheduling algorithms. Our simple algorithm described above comes very close to an optimal schedule calculated by an integer linear program, which runs in 10 minutes. Further, our algorithm is 4 times better than not using a scheduler for 20th percentile of flows, 40% better for 50th percentile and achieves equal total update time. Even if the switches were faster, these results would still hold since the absolute times would change proportionally to the switch processing rate, unless the controller itself becomes the bottleneck.

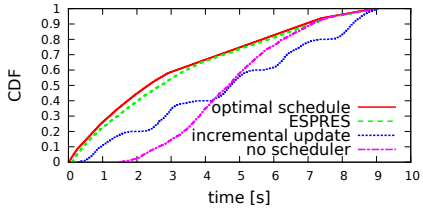

Figure 2: CDF of sub-update times. Our simple schedule is close to optimal and significantly better than having no scheduler.

We also experiment with a scheduler aiming to to minimize rule-space overhead at every switch on a FatTree topology with 20 switches and 370 *sub-updates*; The scheduler orders *sub-updates* such that *sub-updates* deleting rules from overloaded switches are placed first. We find that on average our algorithm causes 10.2% (11.3 rules) maximum overhead,[4] averaged over 6 runs. A naive update without any scheduler results in 76% overhead and an incremental consistent update [3] with 4 rounds causes 16.5% (17 rules) for the same experiment. For comparison, an optimal schedule calculated by integer linear program (assuming constant switch performance) decreases this overhead to 2 rules in our case. However, such computation is impractical as it takes hours to compute even for a moderate number of *sub-updates* (100). We note that the work in [3] and our scheduler are complementary — Katta *et al.* divide a *network update* into sequential rounds, bounding the worst-case overhead within each round. In contrast, ESPRES works on a whole *network update*/round and tries to avoid the worst-case scenario (but without any formal guarantees). Therefore, the two approaches can be combined together.

Acknowledgments. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110. This work was partially supported by the ARC grant 13/18-054 from Communauté française de Belgique.

References

[1] A. Curtis, J. Mogul, J. Tourrilhes, and P. Yalagandula. DevoFlow: Scaling Flow Management for High-Performance Networks. In *SIGCOMM*, 2011.

[2] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *HotSDN*, 2013.

[3] N. P. Katta, J. Rexford, and D. Walker. Incremental Consistent Updates. In *HotSDN*, 2013.

[4] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The Internet Topology Zoo. *Journal on Selected Areas in Communications*, 29(9), 2011.

[5] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz. zUpdate : Updating Data Center Networks with Zero Loss. In *SIGCOMM*, 2013.

[6] R. Mahajan and R. Wattenhofer. On Consistent Updates in Software Defined Networks. In *HotNets*, 2013.

[7] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[3] We estimate the queue length based on the difference between sent Barrier Requests and received Barrier Replies.

[4] The number of extra rules compared to the maximum between initial and final number of rules at each switch